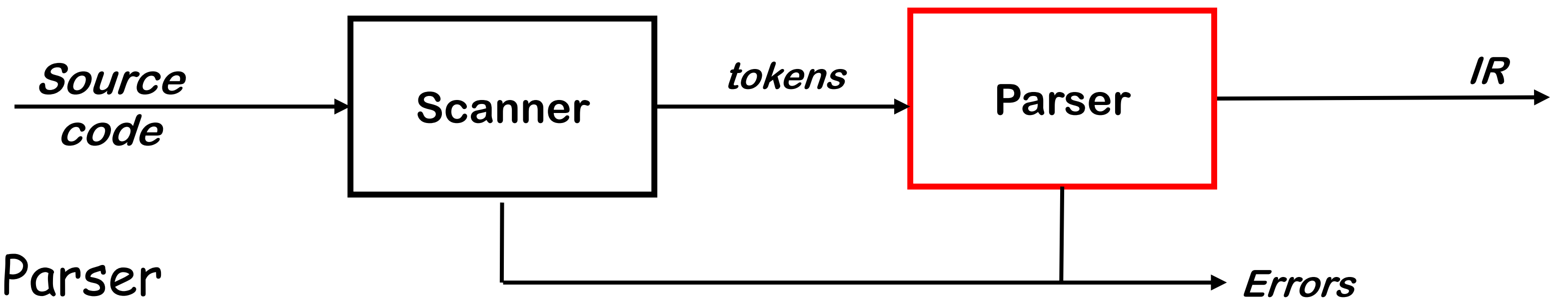
Introduction to Parsing

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

Think of this as the mathematics of diagramming sentences

The Study of Parsing

The process of discovering a *derivation* for some sentence

- Need a mathematical model of syntax — a grammar *G*
- Need an algorithm for testing membership in *L(G)*
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap

1. Context-free grammars and derivations
2. Top-down parsing
   → Hand-coded recursive descent parsers
3. Bottom-up parsing
   → Generated LR(1) parsers

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \textit{SheepNoise}\ \underline{\text{baa}} \\ &\mid\ \underline{\text{baa}} \end{aligned}$$

This *CFG* defines the set of noises sheep normally make

It is written in a variant of Backus–Naur form

Formally, a grammar is a four tuple, $G = (S,N,T,P)$

- S is the *start symbol* *(set of strings in L(G))*
- N is a set of *non-terminal symbols* *(syntactic variables)*
- T is a set of *terminal symbols* *(words)*
- P is a set of *productions* or *rewrite rules* $(P : N \rightarrow (N \cup T)^*)$

Example due to Dr. Scott K. Warren

Deriving Syntax

We can use the *SheepNoise* grammar to create sentences

→ use the productions as *rewriting rules*

Rule	*Sentential Form*
—	*SheepNoise*
2	baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
1	*SheepNoise* baa baa
2	baa baa baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
2	baa baa

And so on ...

While it is cute, this example quickly runs out of intellectual steam ...

A More Useful Grammar

To explore the uses of CFGs,we need a more complex grammar

1	*Expr*	$\rightarrow$	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	$\rightarrow$	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3 ~~2~~	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

x - 2 * y

We denote this derivation: *Expr* $\Rightarrow^*$ id - num * id

- Such a sequence of rewrites is called a *derivation*
- Process of discovering a derivation is called *parsing*

Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of (particular) interest

- *Leftmost derivation* — replace leftmost NT at each step
- *Rightmost derivation* — replace rightmost NT at each step

These are the two *systematic* derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a *leftmost* derivation

- Of course, there is also a *rightmost* derivation
- Interestingly, it turns out to be different

The Two Derivations for x - 2 * y

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

Leftmost derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* <id,y>
6	*Expr* * <id,y>
1	*Expr Op Expr* * <id,y>
2	*Expr Op* <num,2> * <id,y>
5	*Expr* - <num,2> * <id,y>
3	<id,x> - <num,2> * <id,y>

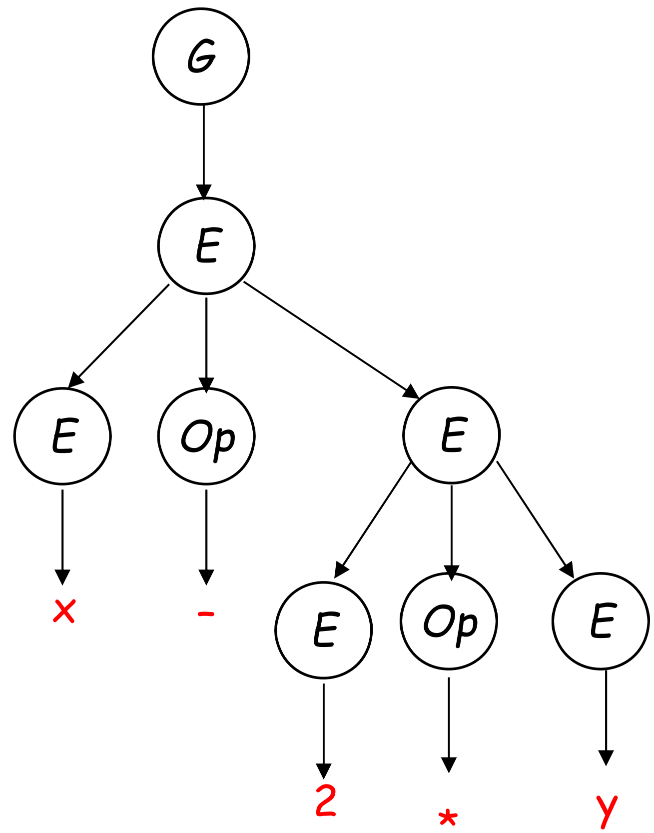
Rightmost derivation

In both cases, *Expr* $\Rightarrow^*$ id - num * id
- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

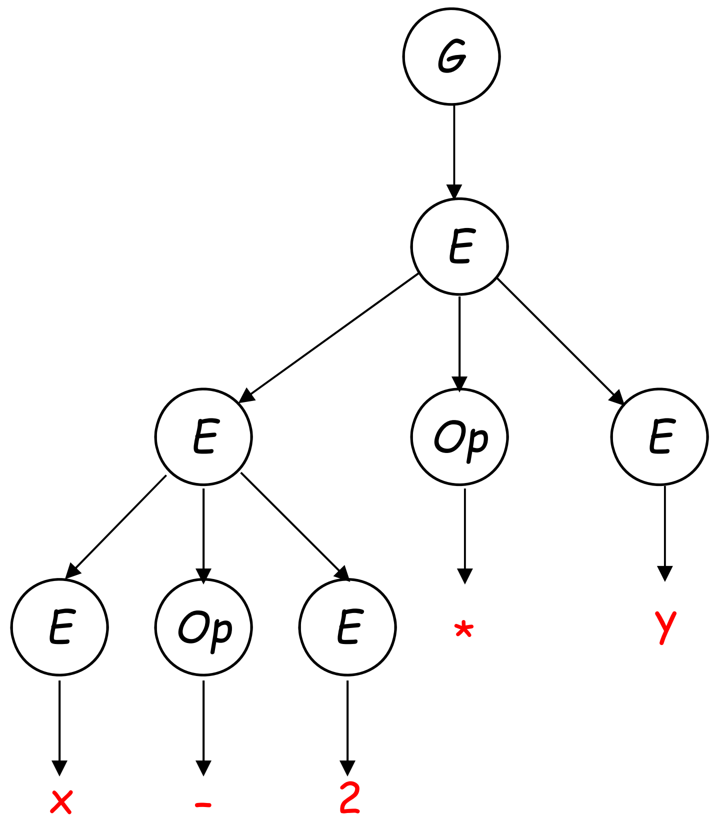


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rightmost derivation

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* <id,y>
6	*Expr* * <id,y>
1	*Expr Op Expr* * <id,y>
2	*Expr Op* <num,2> * <id,y>
5	*Expr* – <num,2> * <id,y>
3	<id,x> – <num,2> * <id,y>



This evaluates as (x – 2) * y

Derivations and Precedence

These two derivations point out a problem with the grammar:
It has no notion of precedence, or implied order of evaluation

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first (*level one*)
- Subtraction and addition, next (*level two*)

Derivations and Precedence

Adding the standard algebraic precedence produces:

<table>
<tr><td></td><td>1</td><td>Goal</td><td>→</td><td>Expr</td></tr>
<tr><td rowspan="3">level two</td><td>2</td><td>Expr</td><td>→</td><td>Expr '+' Term</td></tr>
<tr><td>3</td><td></td><td>|</td><td>Expr '-' Term</td></tr>
<tr><td>4</td><td></td><td>|</td><td>Term</td></tr>
<tr><td rowspan="3">level one</td><td>5</td><td>Term</td><td>→</td><td>Term '*' Factor</td></tr>
<tr><td>6</td><td></td><td>|</td><td>Term '/' Factor</td></tr>
<tr><td>7</td><td></td><td>|</td><td>Factor</td></tr>
<tr><td></td><td>8</td><td>Factor</td><td>→</td><td><u>number</u></td></tr>
<tr><td></td><td>9</td><td></td><td>|</td><td><u>id</u></td></tr>
</table>

Factor → '(' Expr ')'

This grammar is slightly larger

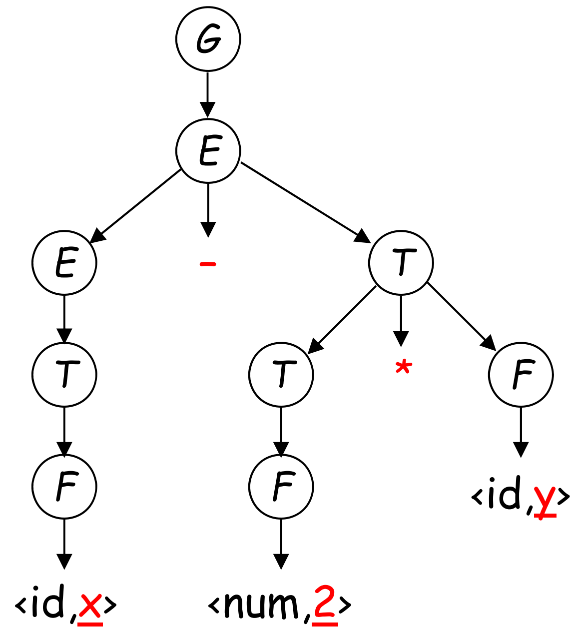
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses $x - 2 * y$

Derivations and Precedence

Rule	*Sentential Form*
—	*Goal*
1	*Expr*
3	*Expr* - *Term*
5	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
7	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
4	*Term* - <num,2> * <id,y>
7	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

The rightmost derivation



Its parse tree

This produces x - (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.

Ambiguous Grammars

x− (y+z)   (x - y) + z

Our original expression grammar had other problems

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
1	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

choice different from the first time

- This grammar allows multiple leftmost derivations for x - 2 * y
- Hard to automate derivation if > 1 choice
- The grammar is *ambiguous*

Two Leftmost Derivations for *x - 2 * y*

The Difference:

- Different productions chosen on the second step

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

Original choice

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
1	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

New choice

- Both derivations succeed in producing *x - 2 * y*

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is *ambiguous*
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the *if*-*then*-*else* problem

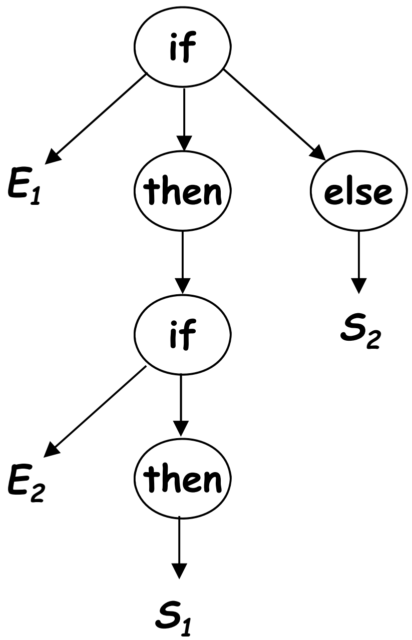
$$
\begin{array}{lll}
\mathit{Stmt} & \rightarrow & \underline{\text{if}}\ \ \mathit{Expr}\ \ \underline{\text{then}}\ \mathit{Stmt} \\
 & | & \underline{\text{if}}\ \ \mathit{Expr}\ \ \underline{\text{then}}\ \mathit{Stmt}\ \ \underline{\text{else}}\ \ \mathit{Stmt} \\
 & | & \ldots\ \mathit{other\ stmts}\ \ldots
\end{array}
$$

This ambiguity is entirely grammatical in nature

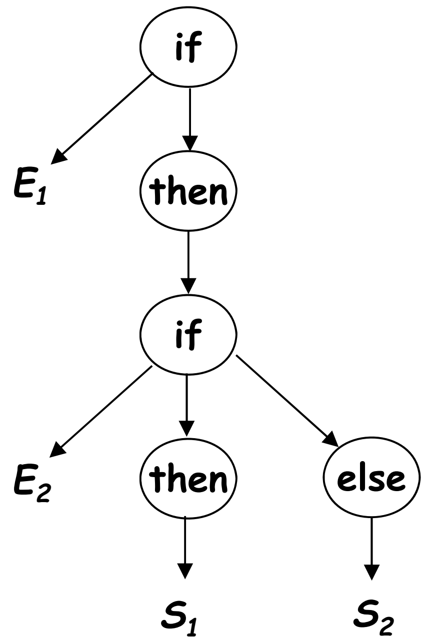
Ambiguity

This sentential form has two derivations

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$



production 2, then production 1



production 1, then production 2

Ambiguity

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if *(common sense rule)*

1	*Stmt*	$\rightarrow$	*WithElse*
2		|	*NoElse*
3	*WithElse*	$\rightarrow$	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	$\rightarrow$	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

Intuition: a *NoElse* always has no else on its last cascaded *else if* statement

With this grammar, the example has only one derivation

Ambiguity

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

Rule	Sentential Form
—	*Stmt*
2	*NoElse*
5	if *Expr* then *Stmt*
?	if E_1 then *Stmt*
1	if E_1 then *WithElse*
3	if E_1 then if *Expr* then *WithElse* else *WithElse*
?	if E_1 then if E_2 then *WithElse* else *WithElse*
4	if E_1 then if E_2 then S_1 else *WithElse*
4	if E_1 then if E_2 then S_1 else S_2

This binds the else controlling S_2 to the inner if

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

a = f(17)

In many Algol-like languages, f could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of *type*, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - → Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (*if*-*then*-*else*)
- Confusion that requires context to resolve (*overloading*)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - → Knowledge of declarations, types, …
  - → Accept a superset of *L(G)* & check it by other means†
  - → This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- → Parsing techniques that "do the right thing"
- → *i.e.*, always select the same derivation

†See Chapter 4